اين مقاله ترجمه اي است از مقاله

Agile Java Development – Refactoring

by Jason Gorman

اصل مقاله از آدرس زير قابل دريافت است:
http://parlezuml.com/tutorials/agilejava/java_refactoring.pdf

Refactoring چيست؟

- بهبود بخشيدن طراحي كدهاي موجود، بدون تغيير كاري كه انجام مي دهند.
- براي اينكه كدها راحت تر قابل تغيير باشند، بايد
  - وابستگي كمتري (loosely coupled) داشته باشند.
  - ماژول ها منسجم تر باشند.
  - كدها قابل فهم تر باشند.
- هر Refactoring بايد ساده و برگشت پذير باشد.
- Automated unit test ها هر مشكلي كه به خاطر refacoring ايجاد مي شود (side effect ها) را نمايان مي سازند.
- اين مفهوم را اولين بار مارتين فاولر در كتاب Reafacoring: Improving the Design of Existing Code معرفي كرد.

Refactoring و روش هاي Agile

- Agile Development بر اين فرض استوار است كه بهترين طراحي پس از تعداد زيادي iteration پديد مي آيد.
- Refactoring براي Agile ضروري است. زيرا با توجه به فيدبك ها و بهبود شناخت ما از اينكه چه چيزهايي در برنامه كار مي كنند و چه چيزهاي نه، الزاما طراحي بايد تغيير نمايد.

مراحل Refactoring (The Refactoring Process)

- يك تغيير كوچك بدهيد، يك Refactoring ساده.
- همه تست ها را انجام دهيد كه اطمينان يابيد همه چيز درست كار مي كنند.
- اگر همه چيز به درستي كار مي كرد، به Refactoring بعدي برويد.
- وگرنه، مشكل را حل كنيد يا كد را به وضعيت قبل از تغيير بازگردانيد. پس الان بايد سيستمي داشته باشيد كه درست كار مي كند.

كدهاي بودار (Code Smells)

كدهايي كه تغيير دادن طراحي شان سخت است عبارتند از:

- كدهاي تكراري (Duplicate Code)
- متدهاي طولاني
- كلاس هاي بزرگ
- Switch statement هاي بزرگ
- Navigation ها و فراخواني هاي تودرتو و طولاني (مثلا (a.b().c().d()
- كنترل هاي فراوان براي null بودن object ها
- داده هاي گروهي (Data clumps). مثل كلاس contact كه شامل فيلدهاي Address، Phone، Email و غيره است. اين مساله را مي توان مشابه جداول غير نرمال در طراحي پايگاه داده رابطه اي دانست.
- كلاس هاي داده (Data Class) . منظور كلاس هايي است كه بيشتر شامل فيلد يا property هستند و متدهاي كمي دارند يا اصلا متد ندارند.
- فيلدهاي Encapsulate نشده مثل متغيرهاي عمومي

انواع Refactoring معمول در جاوا

Extract Class

نگهداري اطلاعات تلفن شخص در كلاس مشتري براساس مدلسازي شيء گرا درست نيست. همچنين اين كار اصل طراحيSingle Responsibility را هم زير پا مي گذارد. از اين رو بدين شكل آن را به دو كلاس جداگانه Refactor مي كنيم.

```
public class Customer
{
	private String name;
	private String workPhoneAreaCode;
	private String workPhoneNumber;
}
```

تبديل مي شود به

```
public class Customer
{
	private String name;
	private Phone workPhone;
}

public class Phone
{
	private String areaCode;
	private String number;
}
```

Extract Interface

برخي كاربران ممكن است بخواهند نام مشتريان را بدانند، در حالي كه بقيه ممكن است فقط نياز به دانستن اين مطلب داشته باشند كه آيا يك object مي تواند به XML، Serialize شود يا خير.
قراردادن toXML() در interface مشتري اصل طراحي Interface Segeration را زير پا مي نهد. اين اصل مي گويد كه بهتر است به جاي يك Interface چند منظوره، چندينinterface خاص داشته باشيم.

```
public class Customer
{
	private String name;

	public String getName(){
		return name;
	}

	public void setName(String string){
		name = string;
	}

	public StringtoXML(){
		return "<Customer><Name>"+ name +"</Name></Customer>";
	}
}
```

تبديل مي شود به

```
public class Customer implements SerializableToXML
{
	private String name;

	public String getName(){
		return name;
	}

	public void setName(String string){
		name = string;
	}

	public StringtoXML(){
		return "<Customer><Name>"+ name +"</Name></Customer>";
	}
}

public interface SerializableToXML{
	public abstract StringtoXML();
}
```

Extract Method

گاهي متدهايي داريم كه چندين كار انجام مي دهند. هرچه حجم كد يك متد، بيشتر باشد فهميدن و اصلاح آن سخت تر است. همچنين منطق موجود در متد در جاي ديگر قابل استفاده مجدد نيست. Refactoring از طريق Extract Method يكي از پركاربردترين روش ها جهت كاهش ميزان تكرار (Duplication) كد ها است.

```
public void PrintAccountDetails(Account account)
{
	// print summary
	System.out.println("Account: "+account.getId());
	System.out.println("Balance: "+account.getBalance());
	// print history
	for (Iterator iterator = account.getTransactions().iterator(); iterator.hasNext();){
		Transaction tx = (Transaction) iterator.next();
		System.out.println("Type: "+tx.getType() +
					" Date: "+tx.getDate().toString() +
					" Amount: "+tx.getAmount().toString());
	}
}
```

تبديل مي شود به

```
public void PrintAccountDetails(Account account)
{
	printSummary(account);
	printHistory(account);
}

private void printSummary(Account account){
	System.out.println("Account: "+account.getId());
	System.out.println("Balance: "+account.getBalance());
}

private void PrintHistory(Account account){
	for (Iterator iterator = account.getTransactions().iterator(); iterator.hasNext();){
		Transaction tx = (Transaction) iterator.next();
		System.out.println("Type: "+tx.getType() +
					" Date: "+tx.getDate().toString() +
					" Amount: "+tx.getAmount().toString());
	}
}
```

Extract Subclass

وقتي يك كلاس، داراي attribute و متدهايي است كه فقط در نمونه هاي (instances) خاص تر، معنـي دارد، مي توانيم يك Subclass از كلاس مربوطه ايجاد كنيم و آن ويژگي ها را در subClass قرار دهيم. اين كار باعث مي شود كه كلاس اصلي بيشتر حالت abstract پيدا كند و اين يكي از نـشانه هـاي طراحـي خوب است.

```
public class Person
{
	private String name;
	private String JobTitle;
}
```

تبديل مي شود به

```
public class Person
{
	protected String name;
}

public class Employee extends Person
{
	private String jobTitle;
}
```

Extract Super-class

وقتي كه دو يا بيش از دو كلاس، برخي خصوصيات (feature) عمومي مشترك دارند، آن عامل اشتراكي را به يك super-class منتقل نماييد. اين كار نتيجه اش abstraction بيشتر و كاهش كدهاي تكراري است.

```
public class Employee
{
        private String name;
        private String jobTitle;
}

public class Student
{
        private String name;
        private Course course;
}
```

تبديل مي شود به

```
public abstract class Person
{
        Protected String name;
}

public class Employee extends Person
{
        private String JobTitle;
}

public class Student extends Person
{
        private Course course;
}
```

Form Template Method

وقتي دو متد در subclass ها مراحل و گام هاي يكساني دارنـد، امـا در هـر مرحلـه كارهـاي متفـاوتي را انجام ميدهند، در اين حالت براي متدها يـك signature ايجـاد كنيـد و متـدهاي اصـلي را بـه كـلاس پايـه منتقل نماييد.

```
public abstract class Party
{
}

public class Person extends Party{

	private String firstName;
	private String lastName;
	private Date dob;
	Private String nationality;

	public void printNameAndDetails()
	{
		System.out.println("Name: "+firstName+" "+lastName);
		System.out.println("DOB: "+dob.toString() +",Nationality: "+nationality);
	}
}

public class Company extends Party{

	private String name;
	private String CompanyType;
	private Date incorporated;

	public void printNameAndDetails()
	{
		System.out.println("Name: "+name+" "+ companyType);
		System.out.println("Incorporated: "+incorporated.toString());
	}
}
```

تبديل مي شود به

```
public abstract class Party {

	public void PrintNameAndDetails(){
		printName();
		printDetails();
	}

	public abstract void printName();
	public abstract void printDetails();
}

public class Person extends Party {

	Private String firstName;
	Private String lastName;
	private Date dob;
	private String nationality;
```

```
	public void printDetails(){
		System.out.println("DOB: "+dob.toString() +",Nationality: "+nationality);
	}

	public void printName() {
		System.out.println("Name: "+firstName+" "+lastName);
	}
}

public class Company extends Party
{
	… etc
}
```

Move Method

اگر يك متد ، از كلاس ديگري بيش از كلاسي كه در آن تعريف شده استفاده مي كند(يا توسط آن استفاده مي شود)، آن متد را به كلاس ديگر منتقل نماييد.

```
public class Student
{
	public boolean isTaking(Course course)
	{
		return(Course.getStudents().contains(this));
	}
}

public class Course
{
	private List Students;

	public List getStudents(){
		return students;
	}
}
```

تبديل مي شود به

```
public class Student
{
}

public class Course
{
	Private List students;

	public boolean isTaking(Student student)
	{
		return students.contains(student);
	}
}
```

مي بينيم كه در حالت refactor شده كلاس student ديگر به Course نيازي ندارد. به اين ترتيب متد isTaking() به داده هايي كه با آن در ارتباط است، نزديك تر شده است.

Encapsulate Field

داده هاي Encapsulate نشده در طراحي شيء گرا، هيچ جايي ندارند. بايد از رويه هاي get و set براي فراهم نمودن دسترسي public يا private به متغيرها استفاده نمود.

```
public class Course
{
        public List students;
}

int classSize = course.students.size();
```

تبديل مي شود به

```
public class Course
{
        public List students;

        public List getStudents() {
                return students;
        }
}

int classSize = course. getStudents().size();
```

براي آشنايي با ساير انواع refactoring به اين آدرس مراجعه نماييد:

http://www.refactoring.com/catalog

ابزارهاي Refactoring جاوا

براي كمك به جابجايي و تغيير نام كدها در جاوا، چندين ابزار وجود دارد. از جمله:

- Jfactor كه با VisualAge و Jbuilder كار مي كند.
- RefactorIt كه يك ابزار پلاگين (plug-in) براي NetBeans، Forte، Jbuilder و Jdeveloper است. همچنين بطور مستقل (standalone) هم كار مي كند.
- Jrefactory كه براي jEdit ، NetBeans، Jbuilder هست و البته بطور مستقل نيز كار مي كند.
- IDE ها از جمله NetBeans و Eclipse اغلب خودشان داراي ابزار و امكانات Refactoring هستند.

شهريور ۸۶

ترجمه: مهيارابراهيمي وفا
لطفا نظرات خود را به mahyar_e_vafa@yahoo.com ارسال نماييد.